Algorithm for Generating Peer-to-Peer Overlay Graphs based on WebRTC Events

Christian von Harscher, Marco Schindler, Johannes Kinzig and Sergej Alekseev
Computer Science and Engineering
Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
{harscher|mschindl|kinzig}@stud.fra-uas.de, alekseev@fb2.fra-uas.de

***Abstract*—The peer-to-peer paradigm is widely used in the distribution of data and documents and direct multimedia communications on the internet. The network nodes, which are concerned in a peer-to-peer network, create an infrastructure that provides and offers a desired functionality or application in a distributed manner. Upcoming communication standards, such as WebRTC, also enable a setup where web browsers can act as peer-to-peer nodes.**

In this paper we present an algorithm for generating connection graphs of the peers based on WebRTC state events. The essential idea of the algorithm is to collect events, generated by changing a signaling and connection state of peers. The algorithm processes the collected events and generates a connection graph of the peers, which represent the overlay topology of the peer-to-peer application.

Additionally we present examples of collected events and corresponding connection graphs.

***Index Terms*—WebRTC, Finite State Machine, Graph, Algorithm**

I. Introduction

Performance monitoring is an challenging aspect of system and service development, it is helpful by detecting and diagnosing performance issues and assists in maintaining a high availability. These days developing teams mainly take data-driven decisions, and these performance measurements play a major role in debugging.

WebRTC is a steadily evolving web communication standard, but it does not provide any interfaces or tools to monitor peer-to-peer overlay structures.

A WebRTC application creates RTCPeerConnection objects [3, sect. 4.3.3] to establish connections between peers. The state of a RTCPeerConnection is represented by three finite state machines: signaling, gathering and connection state machine [1]. Each peer generates an event by changing a signaling, gathering and connection state.

By gathering and storing these events and data it is possible to analyze WebRTC sessions. Since the evaluation of the entire data is a painstaking effort, this article describes the implementation of an algorithm that automatically evaluate these data by generating expressive connection graphs. This contains the representation of the events as a derived graph as well as statistical data of each client. The algorithm is also able to distinguish between the peer itself and the corresponding connections established to other peers, see fig. 1.

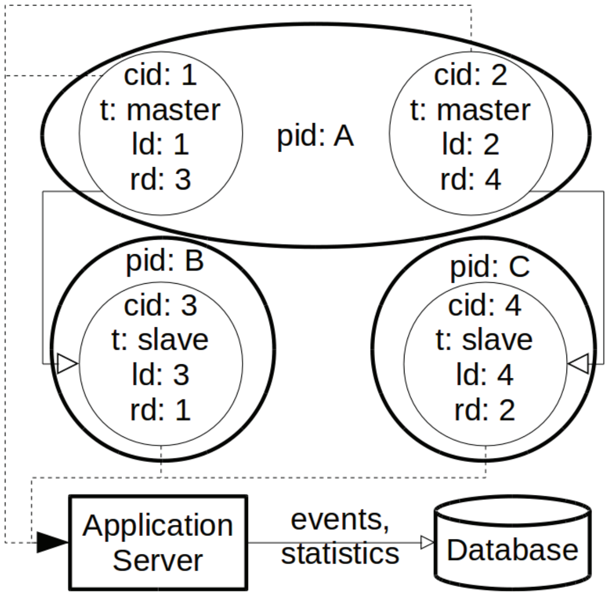

Fig. 1. System Overview, see III-A for details

II. Related Works

Monitoring the performance of P2P based WebRTC applications is a major aspect in system and service environments. Performance can be tracked, issues and non-stable connections can be identified and the diagnosis of these failures help to deliver a high quality of service and availability. Therefore the first step is to gather the WebRTC statistics. Proprietary and also open source frameworks exist for this purpose. One proprietary framework is named *callstats.io* [8] which allows the user to use the vendor's databases to store the WebRTC statistics. The statistics are then represented graphically showing the data rates, network latencies, number of succeeded and failed conferences. Additionally *callstats.io* allows the session users to provide subjective feedback about the call's quality. One drawback is that *callstats.io* is not able to show a graph giving information about the user's connection topology.

One open source method gathering the statistics is presented in [1]. Table I shows an extract of the database table for gathering the WebRTC statistics as presented in [1]. These statistics are important for measuring the general quality of a session. By looking closely at the table it becomes obvious that it is complicated to conclude the quality of a session just from looking at this table. This leads to the basic necessity to have a tool which is able to graphically interpret the gathered data. An algorithm for exactly this purpose is described in this paper.

There are libraries available that provide a way of generating graphs in form of diagrams out of structured information [4], [5] and [6]. This paper will not describe a new way of generating graphs but use Graphviz.

III. Proposed Approach

Basically we are providing an approach for automatically generating a graph structure out of the WebRTC statistics described in [1]. The general aim is to get a structured graph which can be displayed graphically to simplify the forthcoming analysis. Additionally the proposed algorithm can be used to transfer the graph into other data structures (e.g. XML, JSON) for further automated processing.

The following sections describe in more detail the dependency between the WebRTC events to generate the graph, as well as the first steps necessary to implement the algorithm.

A. Terminology and Definition

The system overview is provided in fig. 1. It shows the basic structure with three peers. The peer A is the session initiator which is connected to the peers B and C. The peer-id pid, connection-id cid, type t, local description ld and remote description rd are given. A peer can hold more than one RTCPeerConnection object. The type as well as the corresponding connection-id are defining the direction of a connection, shown as a line with an ending arrow to the slave peer. A connection with type master is always a outgoing connection, whereas a incoming connection is always represented through type slave. Local and remote description are extracted out of the Session Description Protocol (SDP) [7], used for the initialization of the connection.

B. Events Gathering

The event collection is realized by the *WebRTCStateAnalyzer* from [1]. This JavaScript library is used by the WebRTC Application. The client-side code is using the event listeners of the RTCPeerConnection to catch the events. An event E is defined as follows:

$$E(ts, pid, cid, ld, rd, t, l, Z_s) \tag{1}$$

where ts is the timestamp and Z_s the state of an event. The timestamp is necessary to define the ordering of a queue of events Q:

$$\forall E \in Q : ts(E_n) < ts(E_{n+1}) \tag{2}$$

The event listener l is needed for later analysis and is describing which function of the RTCPeerConnection fired the event. The state of the RTCPeerConnection object Z_s is given by $Z_s = (S, G, C)$ where S represents the signaling state machine, G the gathering state machine and C the connection state machine (fig. 4) of the RTCPeerConnection [1, Sec. III].

The actual acquisition of the events is done via a HTTP-POST request from the peer to the application server. After receiving the request the events are stored into a database. In addition statistical data is sent on each event. This data can be consolidated for displaying in the graph. This process and further analysis of the statistical data is described in [2].

C. Algorithm for Generation Connection Graphs

The algorithm creates a directed graph based on gathered raw events. The graph is defined as following:

$$G(N, T, r), r \in N \tag{3}$$

where N are the nodes (peers), T are the transitions (connections) and r is the root peer called session initiator.

Listing 1 shows the main part of the algorithm taking the queue of events Q and returning the graph G as N, T, r (line 10). All events in Q are iterated and added to the node object N. Each peer with its connections is added once in line 3. The second iteration will then loop through all of the node objects for extracting the connections and assigning the transitions. These both algorithms can be found in listings 2 and 3. Last steps to do are to extract all transitions into the transitions object T and finding the root node r. The root node is the node with no incoming transitions.

Listing 1
Main algorithm for generating the graph structure

```
1   GENERATE_GRAPH(Q){
2    for(i = 0; i < sizeof(Q); i++){
3     N.add(Q[i].getPeer())
4    }
5    for(i = 0; i < sizeof(N); i++){
6     GET_CONNECTIONS(Q, &N[i]);
7     ASSIGN_TRANSITIONS(Q, &N[i]);
8     T = N.getAllTransitions()
9     r = N.getRootNode()
10    return N, T, r
11   }
12  }
```

Finding all connections of one node is done by the algorithm in listing 2. For this reason all events in Q need to be iterated. The connection id of the current event *Q[i]* and the given node object *n* are compared. The connection type of *n* needs to be non-slave. After fulfilling these criteria the connection id cid can be added to the given node in line 5.

Listing 2
Algorithm for extracting the connections

```
1   GET_CONNECTIONS(Q, *n){
2    for(i = 0; i < sizeof(Q); i++){
3     if(Q[i].getCid() == n.getCid()
4      && n.getType() != 'slave'){
5      n.addTransitionId(n.getCid())
6     }
7    }
8   }
```

TABLE I
DATABASE EXCERPT WITH WEBRTC RAW EVENTS (COLUMNS EXPLAINED IN III-A)

pid	cid	ld	rd	t	ts	l	S	G	C
18	2364	7938	7768	master	1445330422	oniceconnectionstatechange	stable	complete	connected
58	9854	7768	7938	slave	1445330422	oniceconnectionstatechange	stable	complete	connected
18	3230	3629	7845	master	1445330424	oniceconnectionstatechange	stable	complete	connected
67	3711	7845	3629	slave	1445330424	oniceconnectionstatechange	stable	complete	connected
[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]

The third part of the algorithm in listing 3 is the assignment of the transitions between two peers. A connection is defined by a connection id cid, a local description ld and a remote description rd. The descriptions give the exact information of which node is connected to which other node. The type t is defining the direction of a transition. In this case only the type master is considered so that the first parameter of *assignTransition()* is representing the master and the second parameter represents the slave side of the connection. The transition is directed from the master to the slave node. Additionally the events are treated with descending timestamps, see line 2. Last important thing to remark is that not all events in Q are carrying the local and remote description. To filter the events that are impractical for this use, the constraints in lines 3-5 are defined. The event listener l must be *oniceconnectionstatechange*, the signaling state machine $S \in Z_s$ must be in the state *stable* and the connection state machine $C \in Z_s$ must be in state *connected*. All other events could carry either only one of the descriptions or none of them. Once a transition is found it will be added to the node object in line 9. The local description is always the other peer's remote description and vice versa.

Listing 3
ALGORITHM FOR ASSIGNING THE TRANSITIONS BETWEEN NODES

```
1  ASSIGN_TRANSITIONS(Q, *n){
2   for(i = sizeof(Q); i > 0; i--){
3    if(Q[i].getListener() == '
         ↪ oniceconnectionstatechange'
4     && Q[i].getSig() == 'stable'
5     && Q[i].getCon() == 'connected'
6     && n.getType() == 'master'
7     && Q[i].getLDesc() == n.getRDesc()
8     && transitionNotYetFound()){
9      n.assignTransition(n.getLDesc(), Q[i].
          ↪ getLDesc())
10    }
11   }
12  }
```

D. Example

In this section the whole algorithm will be executed on basis of the presented raw events in table I. These events will be taken as input. The necessary filtering of the events by l, C and S described in III-C is already done. This makes it obvious that some of the events are missing in table I. A balanced tree topology with three nodes is used for the example graph.

After the first loop in line 4 of listing 1 all nodes are extracted:

$$\{18, 58, 67\} \tag{4}$$

The next step is to extract all connections based on the previously extracted nodes. To accomplish this the *GET_CONNECTIONS()* algorithm listing 2 is called. The resulting connections can be represented as:

$$\{(18, 2364), (18, 3230)\} \tag{5}$$

The first value represents the peer id pid and the second value represents the connection id cid. Based on these connections the transitions can be assigned by *ASSIGN_TRANSITIONS* (listing 3). This is the resulting structure containing the cid followed by the local description ld and remote description rd:

$$\{(2364, 7938, 7768), (3230, 3629, 7845)\} \tag{6}$$

Out of this structured data the graph G can be created:

$$G = (\{18, 58, 67\}, \{(18, 58), (18, 67)\}, 18) \tag{7}$$

E. Graph Representation

Now that the necessary graph structure is created by the algorithm it can be visualized in several ways. The fig. 2 represents a graph which is generated from the dot language in listing 4. This generated graph shows a simple representation. It is also possible to use labels for the node itself and its transitions to generated a more complex graph, see fig. 5. The labels include statistical information on the connections between nodes as well as the location of the nodes. Even further analysis can be done by converting the graph structure to other formats like JSON or XML.

Listing 4
EXAMPLE GRAPH IN DOT LANGUAGE

```
digraph G {
  18 -> 58
  18 -> 67
}
```

F. Exception handling

For the analysis and exception handling it is important to detect connecting and disconnecting peers. Fig. 3 shows the joining and leaving peers based on the time. For the forthcoming analysis the following interpretations can be done. Fig. 3 shows a session example where four intervals were identified.

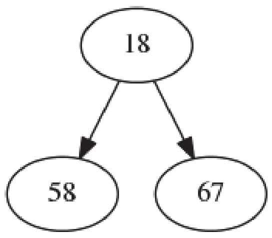

Fig. 2. Example graph with no additional details

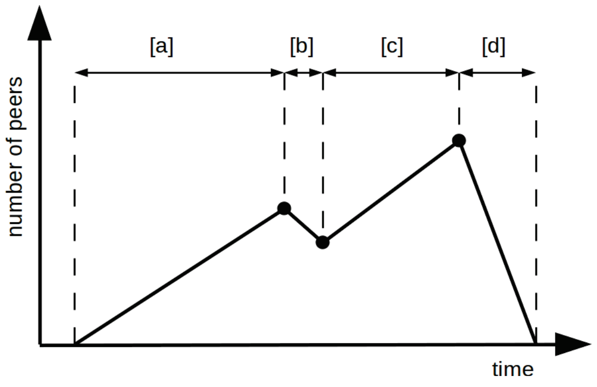

Fig. 3. Connections and disconnections based on time

Interval $[a]$ begins with the opening of a session. Then the number of peers is increasing, the peers join the session. Interval $[b]$ starts when the number of peers is decreasing for the first time. The first peer has disconnected. The reason for a disconnect can be caused by two incidents. Either the user has left the session on purpose or the connection became unstable and this lead to a disconnect. The second described scenario is the most important one for the analysis because it is necessary to distinguish between these two. When the users leaves the session on purpose this must not be shown as an error. When looking at Interval $[c]$ it becomes obvious that the number of peers is increases again. Some peers are again joining the session. Interval $[d]$ is the last identified section. In a short amount of time, the number of clients is decreasing until the number of peers equals zero. This means that the session has ended and every user has left the session.

The connection state machine in fig. 4 is one out of three finite state machines of the RTCPeerConnection. Combined these FSM are representing the status Z_s of a WebRTC connection. With this information it is possible to determine if the connection is successfully established, if there is no signaling or if the connection failed.

IV. Results

It is now possible to extract a expressive graph out of the raw data collected in an WebRTC application. The output

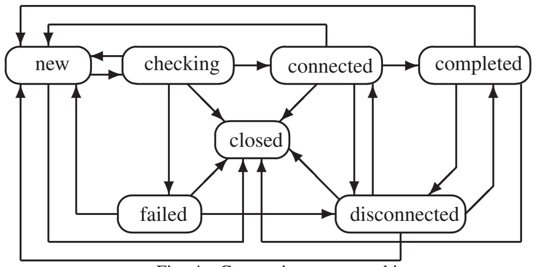

Fig. 4. Connection state machine

graph also contains statistical network data of each connection, determining the link quality. The choosen network statistics for this graph are consisting of the round trip time (RTT) and the send and lost packets between two peers. The algorithm also provides a possibility to evaluate the quality of service through checking of the state machines. There is a distinction between three different states: no signaling, no connection and success. An example graph, representing events and statistical data of three clients and the consequential connections, is shown in fig. 5.

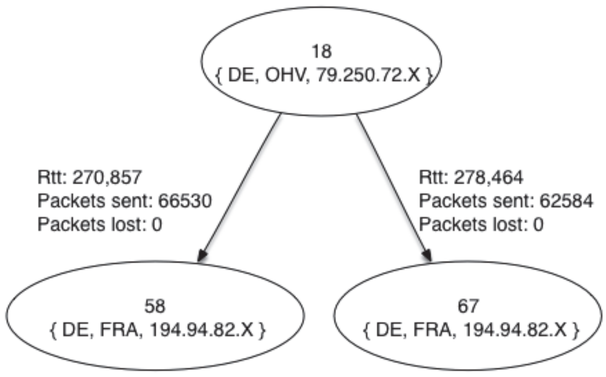

Fig. 5. Example graph with details

V. Conclusion

This article describes an algorithm for generating overlay graphs based on WebRTC state events.

This is a graphical solution to display the evaluation of single WebRTC sessions. Essential information such as the topology of a session or single connection states can be determined by looking at the derived graph.

References

[1] S. Alekseev, C. von Harscher and M. Schindler, Finite State Machine based Flow Analysis for WebRTC Applications, Fourth International Conference on Innovative Computing Technology (IEEE INTECH), University of Bedfordshire, Luton, UK, 2014 *http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6927739*

[2] S. Alekseev, J. Schaefer, A New Algorithm for Construction of a P2P Multicast Hybrid Overlay Tree Based on Topological Distances, The Seventh International Conference on Networks and Communications, 2015

[3] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, WebRTC 1.0: Real-time Communication Between Browsers, Working Draft 10 February 2015 *https://www.w3.org/TR/webrtc/*
[4] AT&T, Graphviz - Graph Visualization Software *http://www.graphviz.org/*
[5] Mathieu Bastian, Eduardo Ramos Ibaez, Mathieu Jacomy, et al., Gephi - The Open Graph Viz Platform *https://gephi.org/*
[6] Wolfram Research, Wolfram Mathematica *http://www.wolfram.com/mathematica/*
[7] M. Handley, V. Jacobson, C. Perkins, SDP: Session Description Protocol *https://tools.ietf.org/html/rfc4566*
[8] Nemu Dialogue Systems Oy, callstats.io, *callstats.io monitors and manages the performance of video calls in an WebRTC application.*, Helsinki, Finland, 01.2016.